

# Likelion Study #AI. Team2

---

Part. Logistic Regression  
2021-10-04



01

## Part 1

1. 이진 분류(Binary Classification)
2. 시그모이드 함수(Sigmoid function)

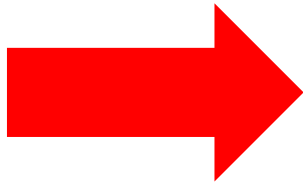
# Contents

## 01. 이진 분류(Binary Classification)

일상 속에는 두 개의 선택지 중 정답을 고르는 문제가 많다.

-> 시험 점수 – 합격 or 불합격

-> 메일 – 정상 메일 or 스팸 메일



둘 중 하나를 결정하는 문제,

**이진분류 (Binary Classification)**

이진 분류를 풀기 위한 대표적인 알고리즘,

**로지스틱 회귀(Logistic Regression)**

# Contents

## 01. 이진 분류(Binary Classification)

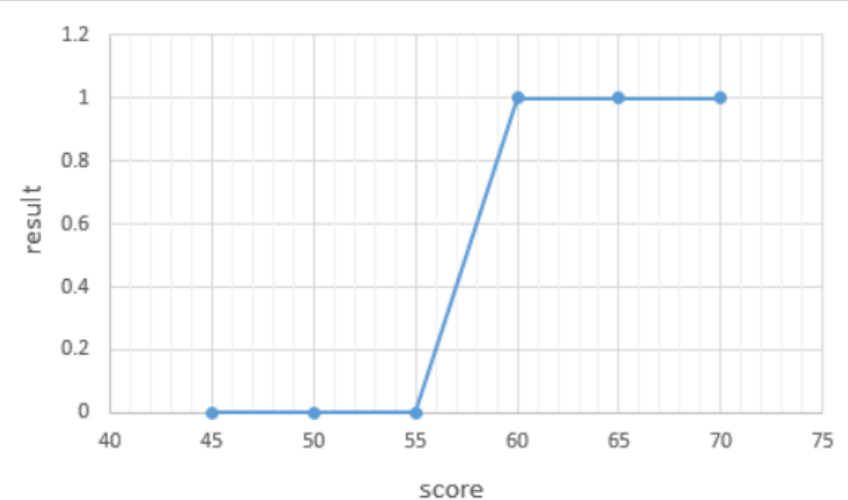
학생들의 시험 성적에 따라  
합격, 불합격이 기재된 데이터

score(x)	result(y)
45	불합격
50	불합격
55	불합격
60	합격
65	합격
70	합격

Score (점수) :  $x$   
Result(결과) :  $y$   
커트라인 공개X

= 0

= 1



**S자 형태의 그래프**

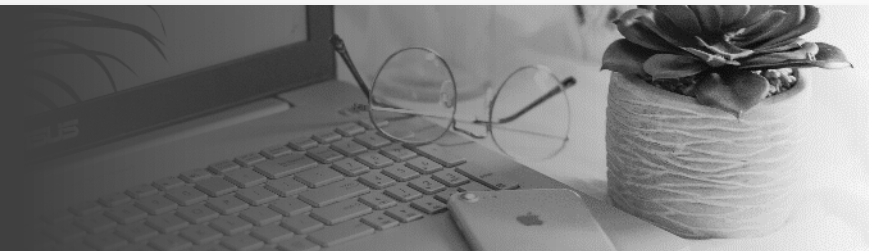
∴ 직선 함수( $Wx+b$ )가 아닌  
S자 형태를 표현하는 함수 필요



**시그모이드 함수**

# Contents

## 02. 시그모이드 함수(Sigmoid function)



: S자 형태를 표현하는 함수

$$H(x) = \text{sigmoid}(Wx + b) = \frac{1}{1 + e^{-(Wx + b)}} = \sigma(Wx + b)$$

선형회귀와 마찬가지로 **최적의 W와 b**를 찾는 것이 목적

Q. W와 b가 함수의 그래프에서 어떤 영향을 주는가?

# Contents

## 02. 시그모이드 함수(Sigmoid function)

### Matplotlib, Numpy импорт

```
%matplotlib inline
import numpy as np # 넘파이 사용
import matplotlib.pyplot as plt # 맷플롯립사용
```

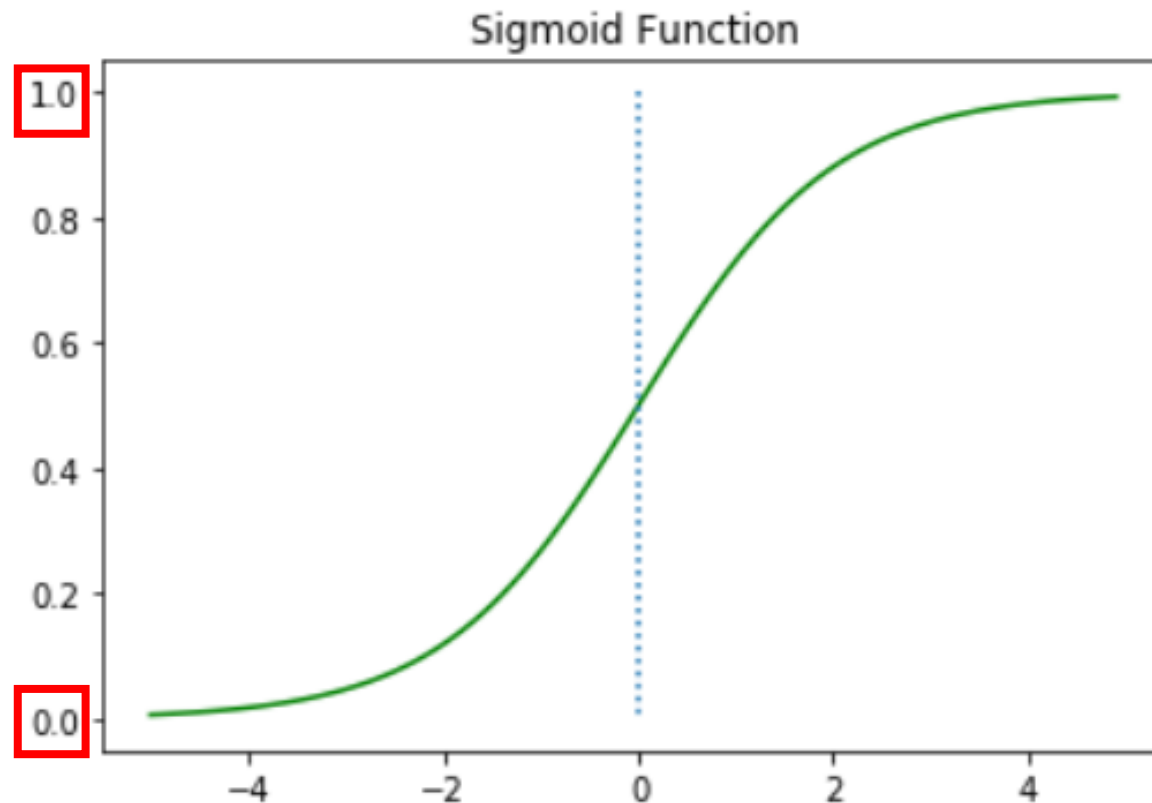
### Numpy 사용하여 시그모이드 함수 정의

```
def sigmoid(x): # 시그모이드 함수 정의
    return 1/(1+np.exp(-x))
```

### W가 1이고 b가 0인 그래프 그리기

```
x = np.arange(-5.0, 5.0, 0.1)
y = sigmoid(x)

plt.plot(x, y, 'g')
plt.plot([0,0],[1.0,0.0], ':') # 가운데 점선 추가
plt.title('Sigmoid Function')
plt.show()
```



출력값은 0과 1 사이

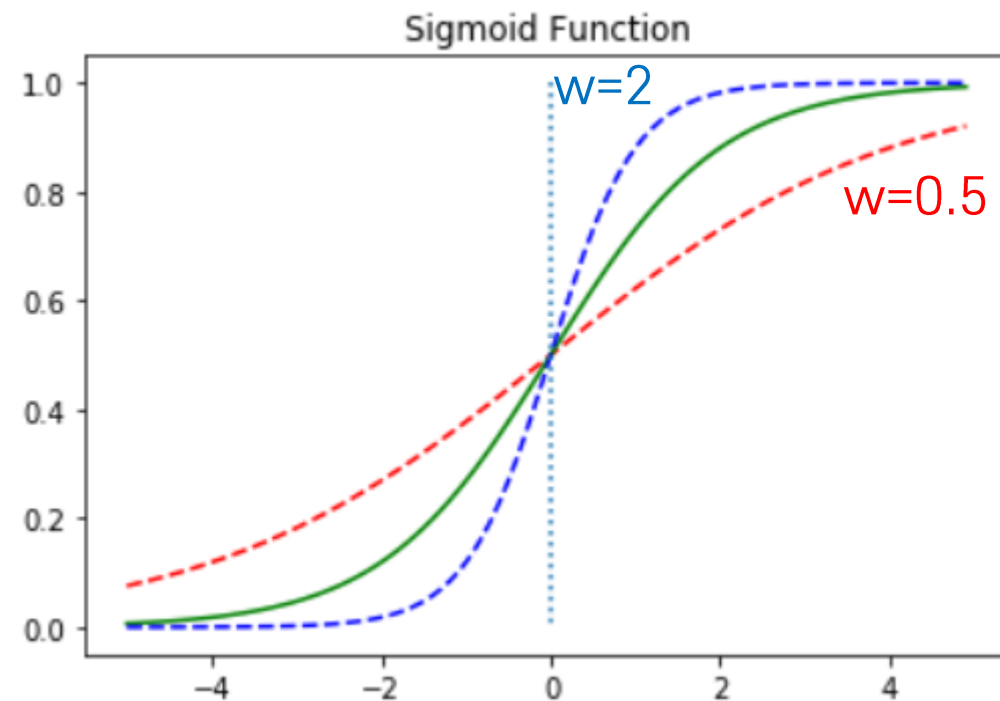
# Contents

## 02. 시그모이드 함수(Sigmoid function)

- W값이 변화될 때 (b는 그대로 0)

```
x = np.arange(-5.0, 5.0, 0.1)
y1 = sigmoid(0.5*x)
y2 = sigmoid(x)
y3 = sigmoid(2*x)

plt.plot(x, y1, 'r', linestyle='--') # w의 값이 0.5일때
plt.plot(x, y2, 'g') # w의 값이 1일때
plt.plot(x, y3, 'b', linestyle='--') # w의 값이 2일때
plt.plot([0,0],[1.0,0.0], ':') # 가운데 점선 추가
plt.title('Sigmoid Function')
plt.show()
```



선형회귀에서는 W는 직선의 기울기 의미했지만  
시그모이드 함수에서는 그래프의 경사도를 의미  
즉, W의 값이 커지면 경사가 커지고 W값이 작아지면 경사가 작아짐



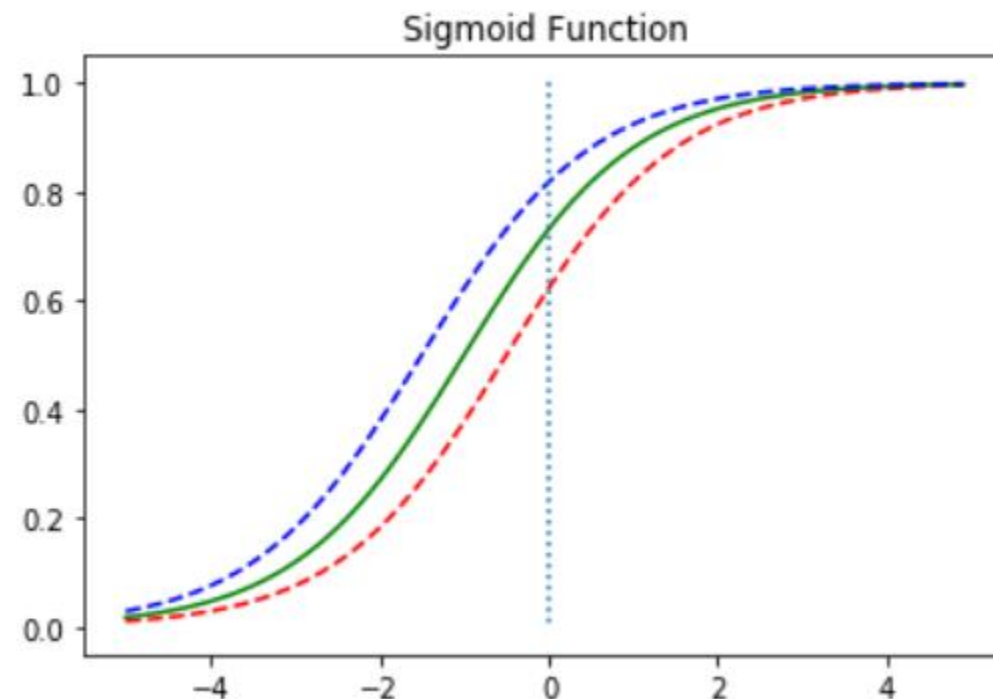
# Contents

## 02. 시그모이드 함수(Sigmoid function)

- b값이 변화될 때 (W는 1)

```
x = np.arange(-5.0, 5.0, 0.1)
y1 = sigmoid(x+0.5)
y2 = sigmoid(x+1)
y3 = sigmoid(x+1.5)

plt.plot(x, y1, 'r', linestyle='--') # x + 0.5
plt.plot(x, y2, 'g') # x + 1
plt.plot(x, y3, 'b', linestyle='--') # x + 1.5
plt.plot([0,0],[1.0,0.0], ':') # 가운데 점선 추가
plt.title('Sigmoid Function')
plt.show()
```

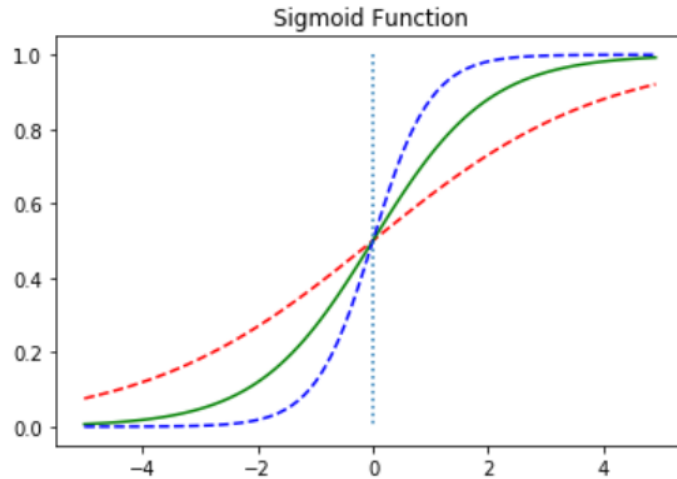


시그모이드 함수에서 b의 값을 달리 줬을 때, 그래프가 좌우로 이동하는 것을 볼 수 있다.



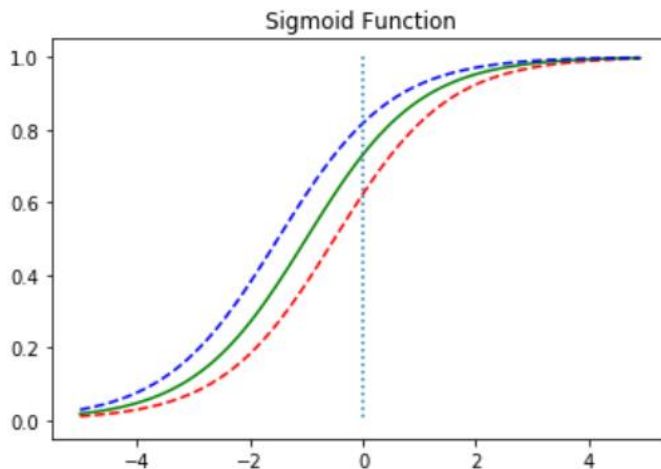
# Contents

## 02. 시그모이드 함수(Sigmoid function)



시그모이드 함수는 입력값이 한없이 커지면 1로 수렴  
한없이 작아지면 0으로 수렴.  
따라서 출력값은 0과 1 사이의 값을 가진다.

### ➡ 분류작업 이용 예시



임계값이 0.5일 때  
출력값이 0.5 이상이면 1(True), 0.5 이하면 0(False)  
확률로 생각하면 50%를 넘겼을 때 해당 레이블이 맞다고 판단,  
넘기지 못했을 때 아니라고 판단한다.



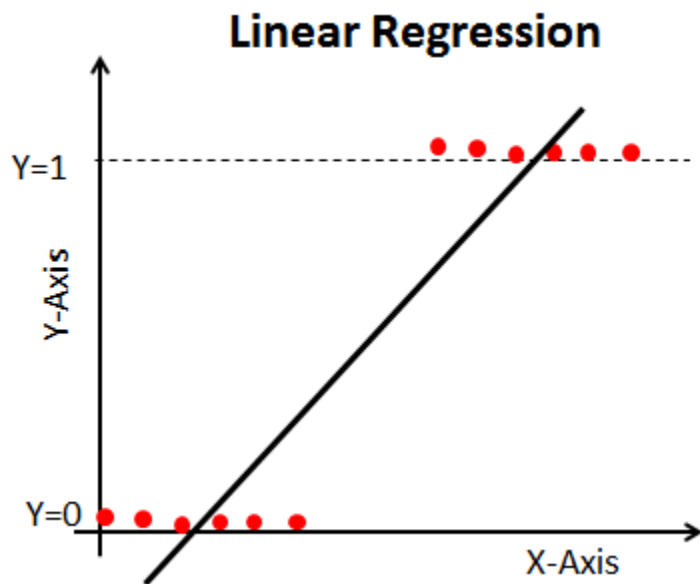
## 02

## Part 2

1. 비용 함수(Cost function)
2. 파이토치로 로지스틱 회귀 구현하기

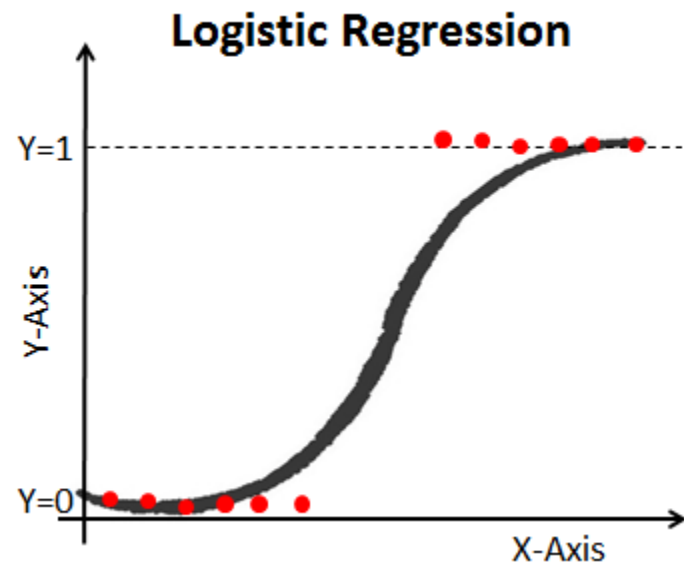
# Contents

## 01. 비용 함수(Cost function)



$$H(x) = Wx + b$$

Linear regression의 Cost 함수

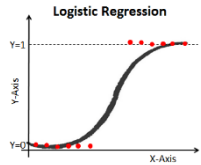


$$H(x) = \text{sigmoid}(Wx + b)$$

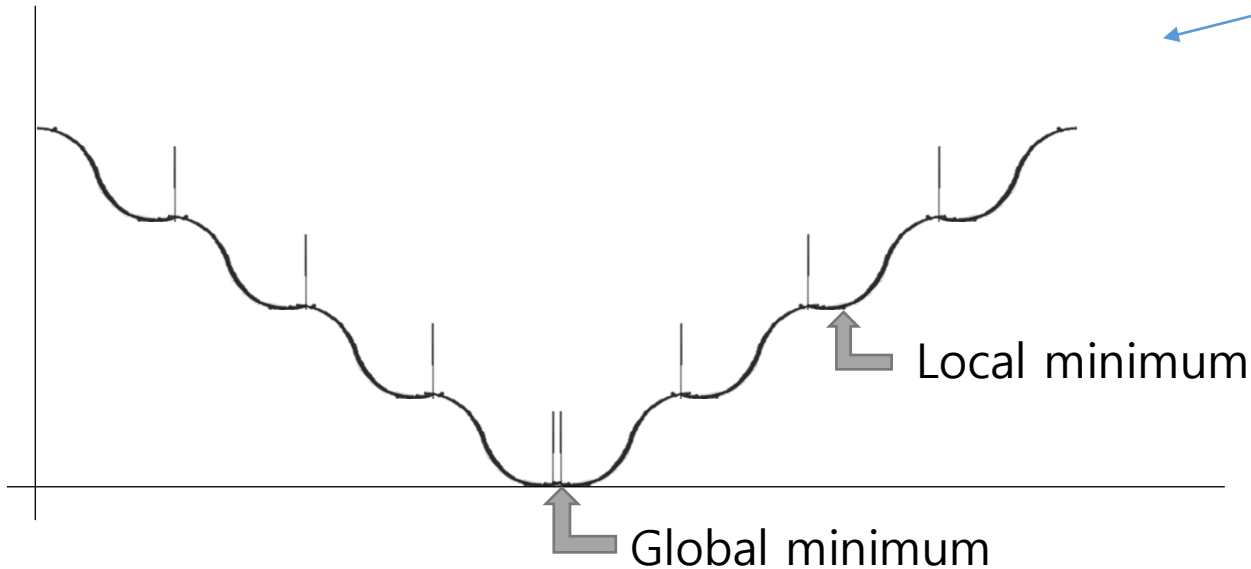
Logistic regression의 Cost 함수

# Contents

## 01. 비용 함수(Cost function)



여러 Sigmoid 함수를 모아 놓은 형태



$$cost(W, b) = \frac{1}{n} \sum_{i=1}^n \left[ y^{(i)} - H(x^{(i)}) \right]^2$$

그래프가 구부러진 형태

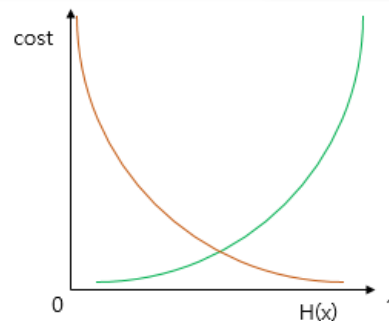
$H(x) = \text{sigmoid}(Wx + b)$  → Sigmoid가 구부러진 형태

시작점에 따라 판단할 수 있는  
최소한의 기울기가 다름.  
따라서, 사용하지 못함.

# Contents

## 01. 비용 함수(Cost function)

$$\text{sigmoid}(Wx + b) = \frac{1}{1 + e^{-(Wx+b)}}$$



제곱의 그래프에 log를 씌워서 구부러진 그래프를 펴주는 느낌

$$\text{if } y = 1 \rightarrow \text{cost}(H(x), y) = -\log(H(x))$$

$$\text{if } y = 0 \rightarrow \text{cost}(H(x), y) = -\log(1 - H(x))$$

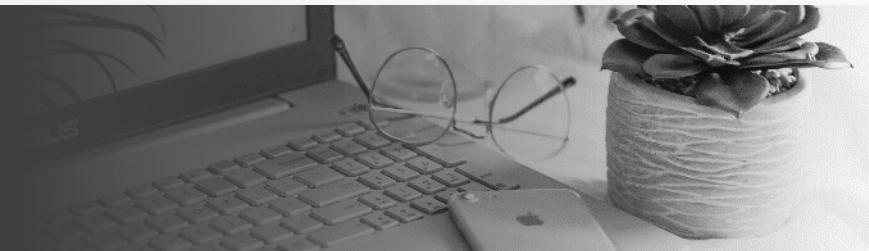
예측 값이 1과 0일 때로 나누어 예측 값과 실제 값의 오차를 구함

$$\text{cost}(H(x), y) = -[y \log H(x) + (1 - y) \log(1 - H(x))]$$

위 공식을 사용하면 한 줄로 표현할 수 있음

# Contents

## 01. 비용 함수(Cost function)



$$cost(W) = -\frac{1}{n} \sum_{i=1}^n [y^{(i)} \log H(x^{(i)}) + (1 - y^{(i)}) \log(1 - H(x^{(i)}))]$$
$$W := W - \alpha \frac{\partial}{\partial W} cost(W)$$

위 비용함수에 대해서 경사 하강법을 적용하여 가중치를 찾는다

# Contents

## 02. 파이토치로 로지스틱 회귀 구현하기

```
x_data = [[1, 2], [2, 3], [3, 1], [4, 3], [5, 3], [6, 2]]
y_data = [[0], [0], [0], [1], [1], [1]]
x_train = torch.FloatTensor(x_data)
y_train = torch.FloatTensor(y_data)
```

```
print(x_train.shape)
print(y_train.shape)
```

```
torch.Size([6, 2])
torch.Size([6, 1])
```

```
W = torch.zeros((2, 1), requires_grad=True) # 크기는 2 x 1
b = torch.zeros(1, requires_grad=True)
```

```
hypothesis = 1 / (1 + torch.exp(-(x_train.matmul(W) + b)))
```

```
print(hypothesis) # 예측값인 H(x) 출력
```

```
tensor([[0.5000],
        [0.5000],
        [0.5000],
        [0.5000],
        [0.5000],
        [0.5000]], grad_fn=<MulBackward>)
```

```
hypothesis = torch.sigmoid(x_train.matmul(W) + b)
```

$$cost(W) = -\frac{1}{n} \sum_{i=1}^n [y^{(i)} \log H(x^{(i)}) + (1 - y^{(i)}) \log(1 - H(x^{(i)}))]$$

```
-(y_train[0] * torch.log(hypothesis[0]) +
  (1 - y_train[0]) * torch.log(1 - hypothesis[0]))
```

```
tensor([0.6931], grad_fn=<NegBackward>)
```

```
losses = -(y_train * torch.log(hypothesis) +
           (1 - y_train) * torch.log(1 - hypothesis))
print(losses)
```

```
tensor([[0.6931],
        [0.6931],
        [0.6931],
        [0.6931],
        [0.6931],
        [0.6931]], grad_fn=<NegBackward>)
```

```
cost = losses.mean()
print(cost)
```

Cost = `tensor(0.6931, grad_fn=<MeanBackward1>)`



# Contents

## 02. 파이토치로 로지스틱 회귀 구현하기

```
F.binary_cross_entropy(hypothesis, y_train)
```

```
tensor(0.6931, grad_fn=<BinaryCrossEntropyBackward>)
```

```
# 모델 초기화
W = torch.zeros((2, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
# optimizer 설정
optimizer = optim.SGD([W, b], lr=1)

nb_epochs = 1000
for epoch in range(nb_epochs + 1):

    # Cost 계산
    hypothesis = torch.sigmoid(x_train.matmul(W) + b)
    cost = -(y_train * torch.log(hypothesis) +
              (1 - y_train) * torch.log(1 - hypothesis)).mean()

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 100번마다 로그 출력
    if epoch % 100 == 0:
        print('Epoch {:4d}/{:} Cost: {:.6f}'.format(
            epoch, nb_epochs, cost.item()
        ))
```

```
hypothesis = torch.sigmoid(x_train.matmul(W) + b)
print(hypothesis)
```

```
tensor([[2.7648e-04],
        [3.1608e-02],
        [3.8977e-02],
        [9.5622e-01],
        [9.9823e-01],
        [9.9969e-01]], grad_fn=<SigmoidBackward>)
```

```
prediction = hypothesis >= torch.FloatTensor([0.5])
print(prediction)
```

```
tensor([[False],
        [False],
        [False],
        [ True],
        [ True],
        [ True]])
```

```
print(W)
print(b)
```

```
tensor([[3.2530],
        [1.5179]], requires_grad=True)
tensor([-14.4819], requires_grad=True)
```



# 03

## Part 3

1. `nn.Module`로 구현하는 로지스틱 회귀
2. 클래스로 파이토치 모델 구현하기

# Contents

## nn.Linear(), nn.sigmoid(), nn.Sequential()

nn.Sequential()

nn.Module() 층을 차례로 쌓을 수 있도록 함.

nn.module()

모든 뉴럴 네트워크 모델의 base class

nn.Linear()

Incomming data에 대해서  $y = Wx + b$ 를 적용.

nn.sigmoid()

Incoming data에 대해서 sigmoid함수를 적용

# SEQUENTIAL

**CLASS** `torch.nn.Sequential(*args)` [SOURCE]

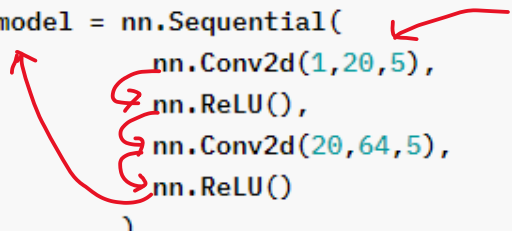
A sequential container. Modules will be added to it in the order they are passed in the constructor. Alternatively, an `OrderedDict` of modules can be passed in. The `forward()` method of `Sequential` accepts any input and forwards it to the first module it contains. It then “chains” outputs to inputs sequentially for each subsequent module, finally returning the output of the last module.

The value a `Sequential` provides over manually calling a sequence of modules is that it allows treating the whole container as a single module, such that performing a transformation on the `Sequential` applies to each of the modules it stores (which are each a registered submodule of the `Sequential`).

What’s the difference between a `Sequential` and a `torch.nn.ModuleList`? A `ModuleList` is exactly what it sounds like—a list for storing `Module`s! On the other hand, the layers in a `Sequential` are connected in a cascading way.

Example:

```
# Using Sequential to create a small model. When 'model' is run,  
# input will first be passed to 'Conv2d(1,20,5)'. The output of  
# 'Conv2d(1,20,5)' will be used as the input to the first  
# 'ReLU'; the output of the first 'ReLU' will become the input  
# for 'Conv2d(20,64,5)'. Finally, the output of  
# 'Conv2d(20,64,5)' will be used as input to the second 'ReLU'  
model = nn.Sequential(  
    nn.Conv2d(1,20,5),  
    nn.ReLU(),  
    nn.Conv2d(20,64,5),  
    nn.ReLU()  
)
```



# Contents

## nn.Sequential()

```
W = torch.zeros((2, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
hypothesis = 1 / (1 + torch.exp((x_train.matmul(W) + b)))
# 1 / (1 + exp(0))
# 1 / 2-
```



```
model = nn.Sequential(
    nn.Linear(2, 1),
    # input_dim = 2, output_dim = 1
    nn.Sigmoid() # 출력은 시그모이드 함수를 거친다
)
```

```
hypothesis = torch.sigmoid(x_train.matmul(W) + b)
```

```
class Sequential(Module):

    def __init__(self, *args):
        super(Sequential, self).__init__()
        if len(args) == 1 and isinstance(args[0], OrderedDict):
            for key, module in args[0].items():
                self.add_module(key, module)
        else:
            for idx, module in enumerate(args):
                self.add_module(str(idx), module)
    def forward(self, input):
        for module in self:
            input = module(input)
        return input
```

# Contents

## nn.Linear()

### LINEAR nn.linear(input size, output size)

**CLASS** torch.nn.Linear(in\_features, out\_features, bias=True, device=None, dtype=None) [SOURCE]

Applies a linear transformation to the incoming data:  $y = xA^T + b$

This module supports `TensorFloat32`.

#### Parameters

- **in\_features** – size of each input sample
- **out\_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

#### Shape:

- Input:  $(N, *, H_{in})$  where  $*$  means any number of additional dimensions and  $H_{in} = \text{in\_features}$
- Output:  $(N, *, H_{out})$  where all but the last dimension are the same shape as the input and  $H_{out} = \text{out\_features}$ .

#### Variables

- **~Linear.weight** – the learnable weights of the module of shape  $(\text{out\_features}, \text{in\_features})$ . The values are initialized from  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ , where  $k = \frac{1}{\text{in\_features}}$
- **~Linear.bias** – the learnable bias of the module of shape  $(\text{out\_features})$ . If `bias` is `True`, the values are initialized from  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$  where  $k = \frac{1}{\text{in\_features}}$

```
def __init__(self, in_features: int, out_features: int, bias: bool = True,
              device=None, dtype=None) -> None:
    factory_kwargs = {'device': device, 'dtype': dtype}
    super(Linear, self).__init__()
    self.in_features = in_features
    self.out_features = out_features
    self.weight = Parameter(torch.empty((out_features, in_features), **factory_kwargs))
    if bias:
        self.bias = Parameter(torch.empty(out_features, **factory_kwargs))
    else:
        self.register_parameter('bias', None)
    self.reset_parameters()

def reset_parameters(self) -> None:
    init.kaiming_uniform_(self.weight, a=math.sqrt(5))
    if self.bias is not None:
        fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        init.uniform_(self.bias, -bound, bound)

def forward(self, input: Tensor) -> Tensor:
    return F.linear(input, self.weight, self.bias)

def extra_repr(self) -> str:
    return 'in_features={}, out_features={}, bias={}'.format(
        self.in_features, self.out_features, self.bias is not None
    )
```

# Contents

## nn.Linear() (cont.)



### TORCH.NN.FUNCTIONAL.LINEAR

```
torch.nn.functional.linear(input, weight, bias=None) [SOURCE]
```

Applies a linear transformation to the incoming data:  $y = xA^T + b$ .

This operator supports **TensorFloat32**.

Shape:

- Input:  $(N, *, in\_features)$   $N$  is the batch size,  $*$  means any number of additional dimensions
- Weight:  $(out\_features, in\_features)$
- Bias:  $(out\_features)$
- Output:  $(N, *, out\_features)$



# Contents

## nn.sigmoid()

### SIGMOID

**CLASS** `torch.nn.Sigmoid` [\[SOURCE\]](#)

Applies the element-wise function:

$$\text{Sigmoid}(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}$$

Shape:

- Input:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input

[\[docs\]](#) **class** `Sigmoid(Module)`:

```
def forward(self, input: Tensor) -> Tensor:  
    return torch.sigmoid(input)
```

hypothesis = torch.sigmoid(x\_train.matmul(W) + b)

```

# optimizer 설정
optimizer = optim.SGD(model.parameters(), lr=1)
#learning rate 1
nb_epochs = 1000
for epoch in range(nb_epochs + 1):

    #W = torch.zeros((2, 1), requires_grad=True)
    #b = torch.zeros(1, requires_grad=True)
    #미분이 가능하게 해준다. (.backward() 메소드를 호출하고 y에 대해서 편미분한 값을 grad 멤버변수에 저장하고 추적이 가능하게 한다)-

    #hypothesis = 1 / (1 + torch.exp(-(x_train.matmul(W) + b)))
    # 1 / (1 + exp(0))
    # 1 / 2-

    #hypothesis = torch.sigmoid(x_train.matmul(W) + b)
    #x_train.matmul(W) + b == > Wx + b
    # H(x) 계산

    model = nn.Sequential(
        nn.Linear(2, 1), # input_dim = 2, output_dim = 1
        nn.Sigmoid() # 출력은 시그모이드 함수를 거친다
    )

    hypothesis = model(x_train)
    # cost 계산

    cost = F.binary_cross_entropy(hypothesis, y_train)

    #cost 함수
    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 20번마다 로그 출력
    if epoch % 10 == 0:
        prediction = hypothesis >= torch.FloatTensor([0.5]) # 예측값이 0.5를 넘으면 True로 간주
        correct_prediction = prediction.float() == y_train # 실제값과 일치하는 경우만 True로 간주
        accuracy = correct_prediction.sum().item() / len(correct_prediction) # 정확도를 계산
        print('Epoch {:4d}/{:4d} Cost: {:.6f} Accuracy {:.2.2f}%'.format(epoch, nb_epochs, cost.item(), accuracy * 100,))

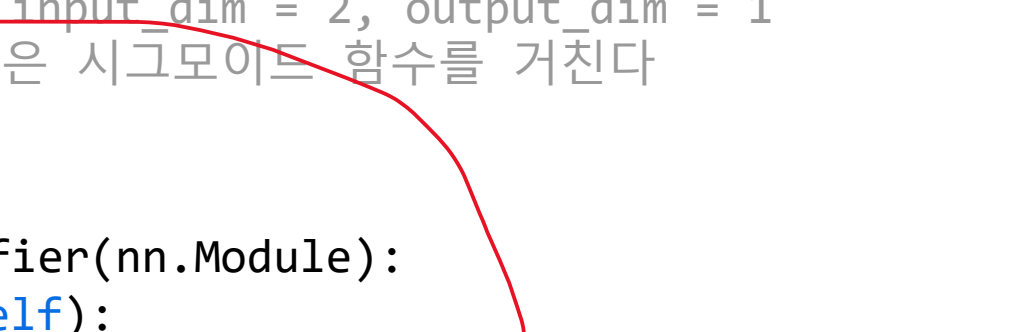
```

# Contents

## Model을 클래스로 구현하기

```
model = nn.Sequential(  
    nn.Linear(2, 1), # input_dim = 2, output_dim = 1  
    nn.Sigmoid() # 출력은 시그모이드 함수를 거친다  
)
```

```
class BinaryClassifier(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.linear = nn.Linear(2, 1)  
        self.sigmoid = nn.Sigmoid()  
    def forward(self, x):  
        return self.sigmoid(self.linear(x))
```



# PYTORCH: CUSTOM NN MODULES

A fully-connected ReLU network with one hidden layer, trained to predict  $y$  from  $x$  by minimizing squared Euclidean distance.

This implementation defines the model as a custom Module subclass. Whenever you want a model more complex than a simple sequence of existing Modules you will need to define your model this way.

```
import torch
```

```
import torch.nn as nn
```

```
class TwoLayerNet(nn.Module):
```

```
    def __init__(self, D_in, H, D_out):
```

```
        """
```

```
        In the constructor we instantiate two nn.Linear modules and assign them as  
        member variables.
```

```
        """
```

```
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)
```

Super().\_\_init\_\_()  
Call base class \_\_init\_\_()

```
    def forward(self, x):
```

```
        """
```

```
        In the forward function we accept a Tensor of input data and we must return  
        a Tensor of output data. We can use Modules defined in the constructor as  
        well as arbitrary operators on Tensors.
```

```
        """
```

```
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred
```

```

class BinaryClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(2, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        return self.sigmoid(self.linear(x))

model = BinaryClassifier()
#객체 생성

# optimizer 설정
optimizer = optim.SGD(model.parameters(), lr=1)
#learning rate 1

nb_epochs = 1000

for epoch in range(nb_epochs + 1):
    hypothesis = model(x_train)

    # cost 계산
    cost = F.binary_cross_entropy(hypothesis, y_train)

    optimizer.zero_grad()
    cost.backward()
    optimizer.step()
# 20번마다 로그 출력
if epoch % 10 == 0:
    prediction = hypothesis >= torch.FloatTensor([0.5]) # 예측값이 0.5를 넘으면 True로 간주
    correct_prediction = prediction.float() == y_train # 실제값과 일치하는 경우만 True로 간주
    accuracy = correct_prediction.sum().item() / len(correct_prediction) # 정확도를 계산
    print('Epoch {:4d}/{:} Cost: {:.6f} Accuracy {:.2f}%'.format(epoch, nb_epochs, cost.item(), accuracy * 100,))

```